共有メモリシステムに対するDPCM 復号処理の並列実装

著者	若谷 彰良
雑誌名	甲南大学紀要.知能情報学編
巻	1
号	1
ページ	119-128
発行年	2008-07-20
URL	http://doi.org/10.14990/00001262

論　文

共有メモリシステムに対するDPCM復号処理の並列実装

若谷彰良
甲南大学　知能情報学部
神戸市東灘区岡本 8-9-1, 658-8501

(受理日 2008 年 6 月 2 日)

概 要

DPCM (Differential Pulse Code Modulation) 符号化はロスレス JPEG 圧縮を含め，さまざまなアプリケーションに広く利用されている．DPCM 復号化は本質的に 1-インデックスもしくは 2-インデックスの回帰演算となっているので，DPCM 復号化は効果的に並列化をすることが難しいものである．しかし，$N \times N$ サイズの画像に対して，$N \times N$ 個もしくは N 個のプロセッサを用いた $(N \log N)$ や $(\log^2 N)$ の計算オーダーの並列アルゴリズムがいくつか知られている．一方，最近のコモディティプロセッサには複数プロセッシングコアを装備されているものもあり，対称型マルチプロセッサ構成をとるパソコンも普及してきている．しかし，その並列度は最大でも 80 程度であり，必ずしも大きいものではない．したがって，$N \times N$ サイズの画像に対して，$N \times N$ 個や N 個のプロセッサを利用することは非現実である．本論文では，$N \times N$ サイズの画像の DPCM 復号化に対して，P 個のプロセッサ ($P << N$) を用いた 2 種類の並列アルゴリズム (*Fat-pipeline* , *P-scheme*) を提案する．実験結果では，いずれも十分な並列効果を示し，6 個のプロセッシングコアを持つプロセッサにおいて，約 3.2 倍のスピードアップを達成した．
キーワード： DPCM, ロスレス JPEG, 並列化, 同期, マルチスレッド , マルチコア

1　はじめに

多数のプロセシングコアを備えたプロセッサは，少ないコストおよび消費電力で高性能計算を実現できる次世代のプラットフォームである．しかし，必ずしも全てのアプリケーションがマルチスレッドを利用したプログラム構成になっているわけではないので，そのようなプラットフォーム上で高性能を実現するためには，アプリケーションの修正もしくは再プログラミングが必要となる．ここでは，DPCM (Differential Pulse Code Modulation) 圧縮，特にその復号化に注目し，マルチコアアーキテクチャを備えた共有メモリシステムに対する並列アルゴリズムの提案と評価を行う．

DPCM はロスレス JPEG [1] を含む多くのアプリケーションで利用されている．ロスレス JPEG はハフマンエントロピー符号化を用いた予測符号化の一種で，表 1 に示す 7 個の予測子が用いられる．ここで，$P_{i,j}$ は，2 次元空間内の (i, j) におけるピクセル値を示す．

ロスレス JPEG の符号化プロセスにおいて，$P_{i,j}$ と予測子との差がハフマン符号化を用いて圧縮され，全ての予測子を用いた圧縮結果を比較し，最少のビット量となる最適な予測子が選択される．ロスレス JPEG の復号化プロセスにおいては，指定された予測子を用いて，圧縮データが伸長される．よって，例えば，予測子 1 を用いて圧縮されている場合，$P_{i,j}$ は $P_{i,j-1}$ が決定された後に，伸長され

る．また，同様に，予測子 2 を用いて圧縮されている場合，$P_{i-1,j}$ が決定された後に $P_{i,j}$ が伸長され，予測子 7 を用いて圧縮されている場合は，$P_{i,j-1}$ と $P_{i-1,j}$ が決定された後に $P_{i,j}$ が伸長される．したがって，復号プロセスにおいては，用いられる予測子によってピクセル間にデータ依存が生じ，並列化を行う際にはこの依存性を保存するようにしなければならない．

表 1: 予測子

pattern	predictor
1	$P_{i,j-1}$
2	$P_{i-1,j}$
3	$P_{i-1,j-1}$
4	$P_{i,j-1}+P_{i-1,j}-P_{i-1,j-1}$
5	$P_{i,j-1}+(P_{i-1,j}-P_{i-1,j-1})*0.5$
6	$P_{i-1,j}+(P_{i,j-1}-P_{i,j-1})*0.5$
7	$(P_{i,j-1}+P_{i-1,j})*0.5$

復号化プロセスにおけるデータ依存によって，2 種類の回帰演算が生じる．データ依存する次元が 1 個の場合，1-インデックス回帰演算がピクセル間で生じ，データ依存する次元が 2 個の場合，2-インデックス回帰演算がピクセル間で生じる．すなわち，予測子 1 および 2 では 1-インデックス回帰演算が生じ，他の予測子では 2-インデックス回帰演算が生じる．1-インデックス回帰演算は，データ依存が生じていない次元を並列化対象とすることにより，比較的容易に並列化が行える．しかし，2-インデックス回帰演算は，並列化対象とする次元が存在しないので，単純な並列化は難しい．

一般に，回帰演算を効果的に並列化することは難しいが，$N\times N$ サイズの画像に対して，$N\times N$ 個もしくは N 個のプロセッサを用いた $(N\log N)$ や $(\log^2 N)$ の計算オーダーの並列アルゴリズムがいくつか知られている [2], [3], [4], [5]．例えば，parallel cyclic reduction では，N 個の 1 次元データに対して，$\log N$ 段の計算で構成され，最初の段では $N-2^0$ 個，次の段では $N-2^1$ 個，さらに次の段では $N-2^2$ 個，最後の段では $N-2^{\log N-1}$ 個のプロセッサが関与する計算となり，$N\times N$ サイズの画像であれば，N 個のプロセッサを用いた計算量 $O(N\log N)$ のアルゴリズムである．

一方，最近のコモディティプロセッサには複数プロセッシングコアを装備されているものもあり，対称型マルチプロセッサ構成をとるパソコンも普及してきている．しかし，その並列度は最大でも 80 程度であり [6]，必ずしも大きいものではない．したがって，$N\times N$ サイズの画像に対して，$N\times N$ 個 や N 個のプロセッサを利用することは非現実である．

本論文では，$N\times N$ サイズの画像の DPCM 復号化に対して，P 個のプロセッサ $(P<<N)$ を用いた 2 種類の並列アルゴリズム (*Fat-pipeline*， *P-scheme*) を提案する．本論文の残りの部分では，次の 2-インデックス回帰演算に注目する．

$$\begin{aligned} x_{i,j} &= a_{i,j}\cdot x_{i-1,j}+b_{i,j}\cdot x_{i,j-1} \\ &+ c_{i,j}\cdot x_{i-1,j-1}+d_{i,j}\ (0<i,j<N) \end{aligned} \tag{1}$$

ここで，$a_{i,j}, b_{i,j}, c_{i,j}$ 及び $d_{i,j}$ はあらかじめ与えられており，i は最内側ループのインデックスとする．

さらに，Pは，独立にスレッドを実行するプロセシングコアの数，Nは，画像の一つの次元のサイズを表す．すなわち，予測子1を用いる場合，復号化プロセスは次のように表される．$P_{i,j}=P_{i,j-1}+D_{i,j}$よって，$x_{i,j}=P_{i,j}, a_{i,j}=0, b_{i,j}=1, c_{i,j}=0, d_{i,j}=D_{i,j}$, である．また，予測子4が用いられる場合の復号化プロセスは，$P_{i,j}=P_{i,j-1}+P_{i-1,j}-P_{i-1,j-1}+D_{i,j}$となり，$x_{i,j}=P_{i,j}, a_{i,j}=1, b_{i,j}=1, c_{i,j}=-1, d_{i,j}=D_{i,j}$である．

2 第1のアプローチ: *Fat pipeline*

式(1)に示すように，$x_{i,j}$は$x_{i-1,j}, x_{i,j-1}$および$x_{i-1,j-1}$に依存する．復号化プロセスを開始する前に，ピクセル$x_{i,j}$は，i-次元に沿って，P個の部分に分割される．すなわち，k-番目のコアは$x_{i,j}$の$i=k*N/P$から$(k+1)*N/P-1$までを担当する．なお，議論を簡単にするために，以降はNはPの倍数とするが，倍数でないケースにも容易に議論を拡張できる．

まず最初に，第1のフェーズにおいて，0-番目のコアは$x_{i,1}$ ($i=0*N/P$から$1*N/P-1$)を復号化する．さらに，第2のフェーズにおいて，0-番目のコアが$x_{i,2}$ ($i=0*N/P$から$1*N/P-1$)を復号化すると同時に，1-番目のコアは$x_{i,1}$ ($i=1*N/P$から$2*N/P-1$)を復号化する．続く第3のフェーズにおいて，0-番目のコアが$x_{i,3}$ ($i=0*N/P$から$1*N/P-1$)を，1-番目のコアが$x_{i,2}$ ($i=1*N/P$から$2*N/P-1$)を復号化すると同時に，2-番目のコアは$x_{i,1}$ ($i=2*N/P$から$3*N/P-1$)を復号化する．よって，P-番目のフェーズ以降は，全てのコアがそれぞれ担当する部分の復号化を並行かつ独立に実行することができ，並列度Pの並列実行を実現することができる．この手法は“pipeline”もしくは“pipelining”と呼ばれる．

```
void lwait(pthread_mutex_t *lmt, pthread_cond_t *lcond, int id, int no){
   pthread_mutex_lock(&lmt[id]);
   while(ldone[id]<lo){
      pthread_cond_wait(&lcond[id],&lmt[id]);
   }
   pthread_mutex_unlock(&lmt[id]);
}
void lstart(pthread_mutex_t *lmt, pthread_cond_t *lcond, int id, int no){
   pthread_mutex_lock(&lmt[id]);
   ldone[id]+=no;
   pthread_cond_signal(&lcond[id]);
   pthread_mutex_unlock(&lmt[id]);
}
```

図 1: 同期関数

しかし，フェーズ間においてそれぞれのプロセシングコアは隣接のプロセッシングコアが前のフェーズの実行を完了していることを確認しなければならないので，フェーズ間での同期が必要となる．したがって，この同期コストがpipelineの並列化効果を下げ，そのオーバーヘッドが実行時間の増加を

引き起こす可能性がある．なお，今回の実装では POSIX 準拠のスレッドライブラリを用いているので，ソフトウェア的な同期のため，オーバーヘッドは比較的大きい．図 1 に示すように，同期のための関数として *lstart* 及び *lwait* 関数を用いた．

pipeline における起動時及び終了時の遅延が問題となる．それぞれのプロセッシングコアはコアの ID 番号と同じ順序でその実行を開始し，終了する．よって，起動時及び終了時の P 個のフェーズにおいては，その並列度は十分ではない．

同期コストを軽減するために，いくつかの行 ($VSIZE$) を一つのブロックにまとめて，ブロック毎に同期を発行する方法 (“Fat pipeline”) を提案する．$VSIZE$ 行毎に同期が発行されるので，同期コストは相対的に $1/VSIZE$ に軽減できる．しかし，それぞれのプロセッシングコアが起動されるまでには通常の pipeline に比べてより多くのタスクの完了を待つ必要があるので，起動がより長く遅延されることになる．Fat pipeline の実行時間は次のように表すことができる．

$$\begin{aligned} T_{fat} &= c_c N(VSIZE + \frac{N - VSIZE}{P}) \\ &+ c_s(P - 1 + \frac{N}{VSIZE} - 1) \end{aligned} \tag{2}$$

ここで c_c はピクセル毎の計算コスト，c_s は同期のコストを表す．

式 (2) から，T_{fat} を最小化する最適な $VSIZE$ は次のように決定できる．

$$VSIZE_{opt} = \sqrt{\frac{c_s}{c_c(1 - \frac{1}{P})}} \tag{3}$$

$$T_{fatopt} = 2N\sqrt{c_c c_s(1 - \frac{1}{P})} + c_c\frac{N^2}{P} + c_s(P - 2) \tag{4}$$

ここで T_{fatopt} は P に関して T_{fat} の最小値である．1 プロセッシングコアでの実行時間は明らかに $c_c N^2$ であるので，T_{fatopt} の $2N\sqrt{c_c c_s(1 - \frac{1}{P})}$ と $c_s(P - 2)$ は，並列化のオーバーヘッド，すなわち，pipeline 遅延と同期コストである．なお，$2N\sqrt{c_c c_s(1 - \frac{1}{P})}$ は N に比例することに注意する．4 章において，実験の結果を用いて Fat pipeline の有効性を確認し，性能と $VSIZE$ の関係を明らかにしていく．

3 第 2 のアプローチ: *P-scheme*

SMP システムにおいて回帰演算を並列に実行する第 2 のアプローチ “P-scheme” を提案する [7], [8]. これは 3 つのフェーズすなわち，*pre-computation* フェーズ，*propagation* フェーズおよび *determination* フェーズから構成される．P-scheme は各行に対して逐次的に適用されるので，2-インデックス回帰演算を次のような 1-インデックス回帰演算に書き直して説明する．

$$w_0 = C \tag{5}$$

$$w_i = s_i \cdot w_{i-1} + t_i, \ (i = 0, 1, ..., N - 1) \tag{6}$$

ここで配列 s, t 及び C はあらかじめ与えられており，配列 w は計算すべき対象とし，各配列は P プロセシングコアがブロック分割して担当する．なお，$N = P \times M + 2$ とする．すなわち，w_i, s_i 及び t_i はそれぞれ $x_{i,j}, a_{i,j}$ 及び $b_{i,j} \cdot x_{i,j-1} + c_{i,j}$ となる．

まず， pre-computation フェーズでは，w_{k*M} が 0 であると仮定し，各プロセシングコア k $(k = 0,1,...,P-1)$ が回帰演算の計算を行い，w_i の計算結果を w'_i として保存する．もちろん，w_{k*M} は必ずしも 0 ではないので，w_i の値を修正しなければならない．ここで $w_i - w'_i$ を Δw_i とする．

$$\Delta w_i \equiv w_i - w'_i = s_i \times (w_{i-1} - w'_{i-1}) \tag{7}$$

$$= \prod_{j=k*M+1}^{i} s_j \times \Delta w_{k*M} \tag{8}$$

したがって，w_{k*M} がコア $k-1$ で決定したとき，コア k 上の配列要素は式 (8) によって真の値に修正することができる．

$$w_i = w'_i + \Delta w_i = w'_i + \alpha_i \cdot w_{k*M} \tag{9}$$

ここで α_i は $\prod_{j=k*M+1}^{i} s_j$ である．

```
/* k is processor ID */
local w', α;
for (j = 1; j < N; j++){
  parallel for k = 0,P-1{ /* 1 */
    w'_0 = 0.0, α_0 = 1.0;
    for(i = 1; i <= M; i++){
      w'_i = a_{k*M+i,j} w'_{i-1} + b_{k*M+i,j} x_{k*M+i,j-1} +
      c_{k*M+i,j};
      α_i = a_{k*M+i,j} × α_{i-1};
    }
  }
  for (k = 1; k < P; k++) /* 2 */
    x_{k*M,j} = w'_M + α_M · x_{(k-1)*M,j};
  }
  parallel for k = 0,P-1{ /* 3 */
    for(i = 1; i <= M; i++)
      x_{k*M+i,j} = w'_i + α_i · x_{k*M,j};
  }
}
```

図 2: P-scheme for DPCM ($x_i = a_i \cdot x_{i-1} + d_i\,(0 < i < N)$, $M = \frac{N}{P}$)

なお，ここで注意しておくことは，w_{k*M} が決定した後には，w_i は任意の順序（昇順，降順，ランダムなど）で修正することができることである．よって，プロセシングコア $k+1$ は w_{k*M+M} の値を必要とするので，propagation フェーズでは，w_{k*M+M} をまず計算し，それをコア $k+1$ に送る．最後の determination フェーズでは，すべてのプロセシングコアは受信した w_{k*M} を用いて，w_{k*M+i} $(i = 1,2,...,M-1)$ を確定する．この P-scheme アプローチを図 2 にまとめる．

P-scheme での実行時間は次のように表される．

$$T_{psc} = N \cdot (c_{pp}\frac{N}{P} + c_{br} + c_{pr}(P-1) + c_{de}\frac{N}{P}) \quad (10)$$

ここで，c_{pp}, c_{pr}, c_{dt} 及び c_{br} は，それぞれ pre-computation, propagation 及び determination フェーズそれぞれのピクセルあたりの計算コスト及びバリア同期のコストを表す．$(c_{pp}+c_{pr}+c_{de})N^2$ がプロセシングコア 1 個の場合の実行時間を表すので，$c_{br} * N$ は並列化のオーバーヘッドコスト，すなわち，バリア同期のコストとなる．なお，バリア同期のコストは P に依存する．

また， Fat-pipeline 及び P-scheme の計算量はそれぞれ $O(N^2/P)+O(N)$ 及び $O(N^2/P)+O(P)$ である．すなわち，Fat-pipeline のオーバーヘッドは理論的には起動時及び終了時の遅延であり，そのコストは $O(N)$ であるのに対し，P-scheme のオーバーヘッドは propagation フェーズに起因し，そのコストは $O(P)$ となる．よって，一般に N は P よりもかなり大きいので，理論的には，P-scheme は Fat-pipeline よりも優れていると考えられる．

4 実験及び考察

両並列アルゴリズムは表 2 に示す SMP システム上で実装された．

表 2: SMP システムの仕様

CPU	Xeon 5335(2.0GHz)
CPU あたりのコア数	4
CPU 数	2
メモリ	4Gbyte
Linux	kernel 2.6.9

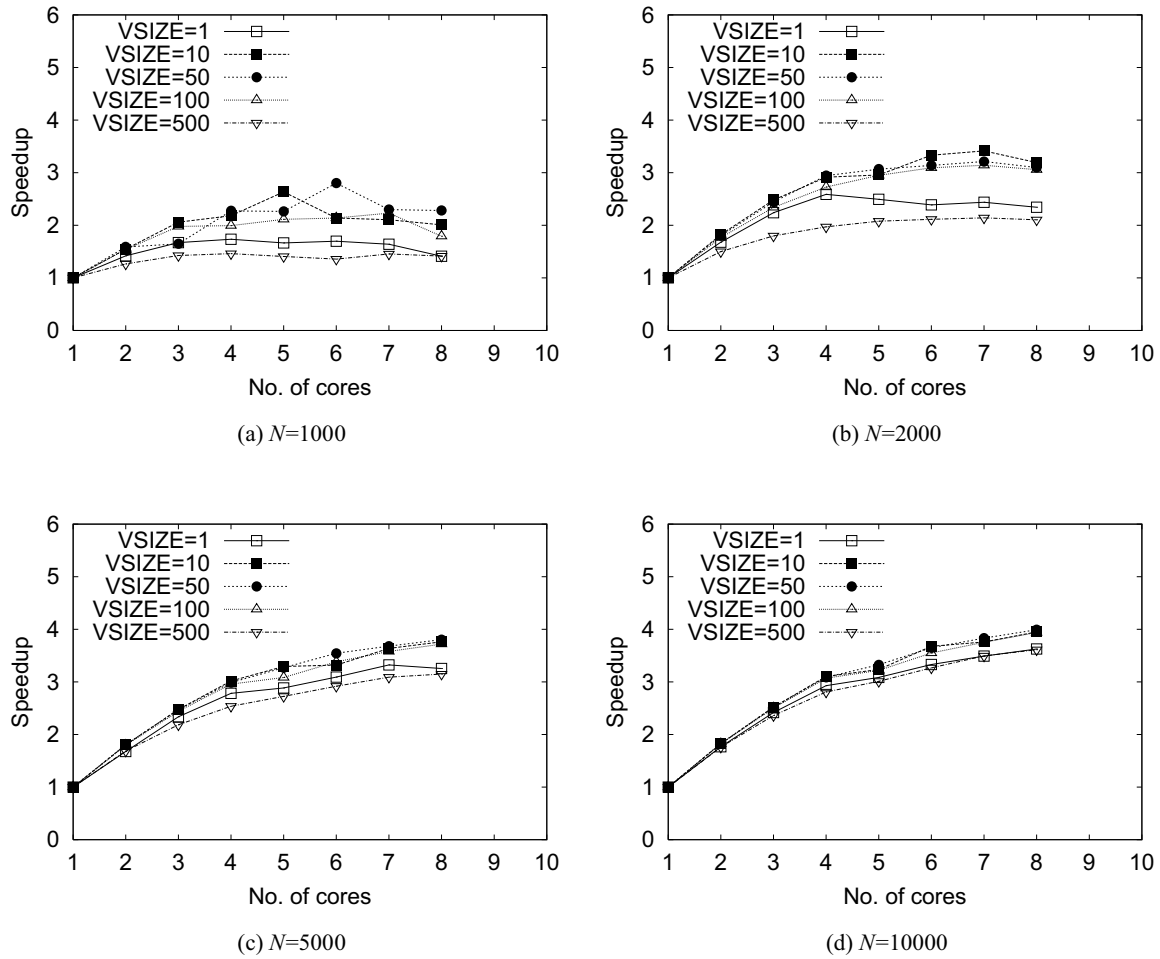
2-インデックス回帰演算を用いた復号化プロセスの実行時間を，問題サイズ (N)，スレッド数，$VSIZE$ などを変えて計測し，並列アルゴリズムのスピードアップを計算する．

4.1 Fat-pipeline

図 3 に，N を 1000, 2000, 5000 及び 10000 にしたときの Fat-pipeline の結果を示す．問題サイズが増加するにつれて，$VSIZE$ とは関係なくに並列度は増加する．よって 5000 及び 10000 のケースではより高いスピードアップが達成できる．しかし，1000 及び 2000 のケースでは，アルゴリズムの有効性は $VSIZE$ に依存する．

問題サイズが 2000 の場合，プロセッシングコアを増加させると，スピードアップも増加するが，その効果も 6 コアで飽和する．例えば，$VSIZE$ を 500 にすると，コア数を 5, 6, 7, 8 にしたときのスピードアップはそれぞれ 2.08, 2.11, 2.14, 2.10 である．前の章で述べたように，Fat-pipeline は，起動・終了時の遅延および隣接コア間の同期コストによって，その有効性が減少しうる．すなわち，$VSIZE$ が非常に大きいと，起動・終了時の遅延も大きくなり，一方，$VSIZE$ が非常に小さいと，同期コストが

相対的に大きくなる．したがって，*VSIZE* を適切かつ最適に選択することによって，より大きいスピードアップを達成することができる．すなわち，8 コアの場合は，*VSIZE* を 10 にすることにより，3.20 のスピードアップが達成できる．



(a) *N*=1000　(b) *N*=2000

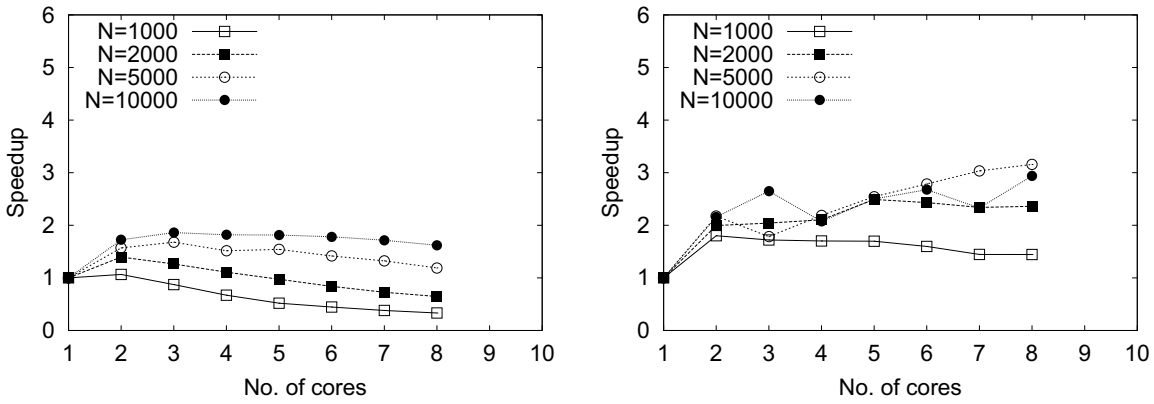
(c) *N*=5000　(d) *N*=10000

図 3: Fat Pipeline の有効性 (*N*=2000)

4.2 P-scheme

図 4 の左図は POSIX スレッドライブラリを用いて問題サイズを変えて計測したときの P-scheme のスピードアップを示している．しかし，*N* が 10000 のときでも，最大のスピードアップは 2.0 前後であり，例えば，コアを 2 もしくは 3 にしたときはそれぞれ 1.74　および 1.86 となっている．他の場合もこれよりも悪い結果となる．P-scheme の計算量は $O(N)$ であり，スケーラブルに並列化可能であるが，バリア同期のコストが比較的大きく，全体の性能に大きく影響を与えている．

最初の実験では，バリア同期を “pthread_cond_broadcast” と “pthread_cond_wait” を組み合わせて実現していたが，バリア同期は $O(N)$ のオーダーで頻繁に実行されるので，その同期コストがアルゴリズム全体の有効性を下げる要因となっていた．



(a) slow 同期実装　　(b) fast 同期実装

図 4: P-scheme の有効性

図 4 の右図は簡易なバリア同期の実装を用いた場合のスピードアップを示す．実装にはスレッド間で共有変数をカウントアップするだけで制御するようにした．

例えば，N が 50000 のとき，コア数が 5, 6, 7, 8 の場合のスピードアップは，それぞれ 2.54, 2.79, 3.03 および 3.16 となる．よって，バリア同期の実装を軽く，そのコストを軽減することにより，P-scheme の有効性は Fat-pipeline のものとほぼ同程度になる．

4.3 考察

3 章の最後で述べたように，計算量評価では P-scheme のほうが Fat-pipeline よりも優れている結果であった．しかし，今回の実験では，P-scheme のバリア同期のコストを若干トリッキーな手法で下げることによって，ほぼ同等の性能にすることができたが，Fat-pipeline を上回るまでには至らなかった．これにはいくつかの理由が考えられる．

まず第 1 の理由としては，P-scheme アプローチ自体の計算量評価において，係数部分を考慮しなかったことがあげられる．すなわち，P-scheme アプローチは，元々の回帰演算よりも計算量が増えている．例えば，pre-computation フェーズで行う計算と determination フェーズで行う計算にはほぼ同等のものが含まれ，元の回帰演算の 2 倍近い計算が含まれることになる．したがって，最適に実装された場合でも，P 個のコアに対して $P/2$ 程度のスピードアップが上限となる．

また，第 2 の理由としては，Fat-pipeline のオーバーヘッド部分が $O(N)$ であり，P-scheme のオーバーヘッド部分が $O(P)$ であるので，今の低並列度 ($P = 8$) の環境であれば，問題サイズをさらに大きくすると，Fat-Pipeline の優位性がくずれ，P-scheme アプローチの有効性がより一層高まると考えら

れる．

さらに P-scheme の性能を向上するためには，各行を逐次に実行する代わりに，複数行が扱えるようにアルゴリズムの変更を行い，バリア同期のコストを相対的に下げることが有効であろうと考えられる．

5 おわりに

DPCM 符号化はロスレス JPEG 圧縮を含む幅広いアプリケーションで利用されており，そのアルゴリズムは本質的に 1-インデックスもしくは 2-インデックスの回帰演算である．本論文では，$N \times N$ サイズの画像に対して，P プロセッシングコア ($P << N$) で実行できる並列アルゴリズム Fat-pipeline と P-scheme を提案した．

我々の実験では，2000×2000 サイズの画像に対する Fat-pipeline のスピードアップは $VSIZE = 1$ のときに比べて大幅に改善できた．解析的及び実験的な評価により，マルチコアプロセッサにおいては，最適な $VSIZE$ を決定することが，実行時間を最小化するキーとなることが分かった．

一方，P-scheme を用いた予備的な実験では，バリア同期をより軽量な実装に変更することにより，その実行時間が改善でき，Fat-pipeline とほぼ同等なスピードアップが達成できた．

謝辞

本研究の一部は，文部科学省オープン・リサーチ・センター整備事業「知的情報ネットワークによる地域密着型教育の高度情報化に関する研究」(2004-2008) の支援を受けて，実施されました．

参考文献

[1] H. Kongli, “Experiments with a lossless JPEG Codec,” *Thesis Cornell Univ.*, 1994.

[2] A. Youssef, “Parallel algorithms for multi-indexed recurrence relations with applications to DPCM image compression,” in *Proc. Data Compression Conference*, p. 584, 1998.

[3] R. Hockney, *Parallel Computer 2* , Adam Hilger, 1988.

[4] J. Lopez and E. Zapata, “Unified architecture for divide and conquer based tridiagonal system solver,” *IEEE trans. on Computer*, vol. 43, no. 12, pp. 1413-1425, 1994.

[5] E. Dekker, and L. Dekker, “Parallel minimal norm method for tridiagonal linear systems,” *IEEE trans. on Computer*, vol. 44, no. 7, pp. 942-946, 1995.

[6] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote and N. Borkar, “An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS,” in *Proc. ISSCC2007*, p. 98, 2007.

[7] A. Wakatani, “A parallel scheme for solving a tridiagonal matrix with pre-propagation,” in *Proc. of 10th Euro PVM/MPI Conference*, pp. 222-226, 2003.

[8] A. Wakatani, “A parallel and scalable algorithm for calculating linear and non-linear recurrence equations,” in *Proc. Int’l Conf. Parallel and Distributed Computing and Networks*, pp. 446-451, 2004.